AFRL-RI-RS-TR-2017-223

MOKA WITH RISTRETTO

ASSURED INFORMATION SECURITY, INC.

NOVEMBER 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2017-223 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* NOVEMBER 2017	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* DEC 2015 – MAY 2017

4. TITLE AND SUBTITLE

MOKA WITH RISTRETTO

5a. CONTRACT NUMBER FA8750-16-C-0057

5b. GRANT NUMBER N/A

5c. PROGRAM ELEMENT NUMBER 61101E

6. AUTHOR(S)

Mark Bridgman, Jonathan Miodownik

5d. PROJECT NUMBER STAC

5e. TASK NUMBER MO

5f. WORK UNIT NUMBER KA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Assured Information Security, Inc.
153 Brooks Road
Rome, NY 13441

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER

AFRL-RI-RS-TR-2017-223

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Moka explored the prevalence of a sub-Turing Complete (sub-TC) program semantics in a large-scale codebase. It showed that the high concentration of code can be implemented without the full power of a Turing machine, justifing the utility of restricted execution environments, and their use in code development. In addition, the Ristretto ECP to Moka investigated the feasibility of automatically restricting provably-bounded software components to sub-TC execution environments for enhanced security of legacy code-bases. It used compiler and Just-in-Time (JIT) loop unrolling, combined with modern CPU branch tracing functions to automatically enforce forward only execution on applicable portions of a given program.

15. SUBJECT TERMS

Language Theoretic Security, LangSec, Intel Processor Tracing, Forward Only Execution

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **MARK K. WILLIAMS**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	25	**19b. TELEPHONE NUMBER (*Include area code*)** **N/A**

Table of Contents

1.0 SUMMARY 1
2.0 INTRODUCTION 2
2.1 Problem Statement 2
2.2 Background 3
2.2.1 Weird Instructions and Weird Machines 3
2.2.2 Turing Completeness and Security Vulnerabilities 3
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES 5
3.1 The Moka Analysis System 5
3.1.1 Motivation 5
3.1.2 Moka Implementation 6
3.1.3 Moka’s LLVM Passes and Utility Code 6
3.1.3.1 Cycle Pass 6
3.1.3.2 Recursion Pass 7
3.1.3.3 Loop Count Pass 8
3.1.3.4 Function Count 9
3.1.3.5 Source Lines of Code 9
3.1.3.6 Loop Graph 10
3.2 Ristretto 10
3.2.1 Motivation 10
3.2.2 Implementation 10
4.0 RESULTS AND DISCUSSION 11
4.1 Moka Pass Results 12
4.1.1 Interpretation of Results 13
4.2 Ristretto Results 14
4.2.1 Ropper 14
4.2.2 Performance Impact 14
4.2.2.1 Intel Processor Tracing 14
4.2.2.2 Forward-Only Enforcement 15
5.0 CONCLUSION 16
6.0 REFERENCES 17
7.0 LIST OF ACRONYMS 20

List of Figures

Figure 1: Computational Machines by Complexity Class 4
Figure 2: Example of Call Cycle 7
Figure 3: The JSON output from the cycle pass has two cycles: cycle_ref_1 and cycle_ref_2. 7
Figure 4: Recursive Function Example Code 8
Figure 5: LLVM IR for a simple loop 8
Figure 6: Unresolved Upper Bound 9
Figure 7: Promotion Pass 9
Figure 8: Graphical Representation of Non-Terminating Loops in Linux Kernel 12
Figure 9: Power Series for Approximating Pi 15

List of Tables

Table 1: Loop Classification 12
Table 2: Notable Regions of Non-termination 13
Table 3: Source Lines of Code by Source Language 13
Table 4: Intel PT Runtime Overhead 15

1.0 SUMMARY

Moka was a research effort that explored the prevalence of a sub-Turing Complete (sub-TC) program semantics on a large-scale codebase. By demonstrating the high concentration of broadly used and critical code that can be implemented without the full computational power of a Turing machine via exploration of the Linux Kernel codebase, Moka justifies the utility of using restricted execution environments, and by extension, warrants their use in code development. If these regions can be restricted, *weird machines*—the unintended execution environments created when data is not explicitly handled or formally parsed—can be reduced or eliminated, resulting in programs which are more likely to execute per their author's intentions.

Moka explores two main themes. First, the path explosion problem is described and why it is an impediment to formally reasoning about program behavior, and explain how a less expressive computational model can reduce the number of symbolic execution paths dramatically. Moka's technical details are then described and results are analyzed within the constraints of the two major questions it asks: 1) How reliably can we detect the computational power of the machine representing a piece of code? and 2) what percentage of a large existing codebase can be categorized as a sub-TC machine, and by this, leverage the security benefits of language-theoretic security (LangSec)?

A set of metrics-gathering tools were developed using the LLVM compiler framework and custom python to determine the prevalence of code that can be executed in a sub-TC environment. By detecting cycles in function call graphs, analyzing how and if loops terminate, and analyzing control-flow-interacting constructs such as spin locks, Moka quantified the fully TC code in the Linux kernel, and consequently makes the argument that the kernel's constituent code overwhelmingly resides in the sub-TC domain. Via this demonstration, Moka makes the case that LangSec principles are not only useful in theory, but practical in application. It should be noted that while the Moka effort inspected the Linux kernel, the tools produced during the course of the effort are sufficiently general as to be applicable to arbitrary programs written in the C language.

The Ristretto ECP was an addition to the Moka project that sought to take advantage of the fact that most programming tasks do not require the full power offered by a TC runtime environment. This extended effort explored the feasibility of applying hardware-enforced bounds on sub-TC program components in an automated fashion, creating an effectively-sub-TC runtime environment. Through the novel application of Intel Processor Tracing and LLVM's just-in-time compilation, legacy programs can benefit from greater protections against exploitation. Ristretto aimed to give developers the programming tools to make development of safer and more secure software easier and automated analysis problems easier.

2.0 INTRODUCTION

2.1 Problem Statement

A fundamental component of most computer security research is the identification and study of exploitation. An exploit is the reimagining of code written for a purpose other than the intended one of its author, often with malicious intent. While there are many simple examples of this, such as buffer-overflow based code injection attacks, an increasing number of software and hardware defense mechanisms has shifted the landscape towards more sophisticated attacks such as Return Oriented Programming attacks (ROP). ROP attacks manipulate the return addresses stored on a program's call stack, effectively chaining together preexisting code segments to create paths of execution that can subvert many conventional security defenses; these attacks are accordingly more difficult to defend against.

Many of the efforts to secure software have been taken from a top-down perspective; concentrating on correct ("bug-free") specification of developed systems. One method of accomplishing this is determining how an input to the computing system propagates and affects the system. By systematically demonstrating that for any given input there is a set of acceptable and completely known set of outcomes, one can symbolically prove the trustworthiness of a system. However, when attempting this under normal circumstances, one encounters an asymmetric relationship between the inputs being mapped and the number of possible outcomes that can result known as the *path explosion problem* or *state explosion problem*. More precisely, the linear increase of inputs map to an exponential growth of possible paths of execution a program can take. As a result, the analysis of execution paths is intractable for most non-trivial programs.

The LangSec field applies a different perspective to software security, reasoning that the cause of many software vulnerabilities stems from the lack of formal language that specifies valid or expected inputs to software sections. Supplying crafted inputs to a program that does not restrict the data it consumes can drive the program's execution into unexpected states through unexpected computation [1]. The verification can be attained by writing the input parsing code to reject any inputs that fall outside the formal language specification (grammar) of valid or expected inputs. Using such a language would allow programmers to build software components that are constructed as verifiable recognizers [2].

When designing software, often developers will make use of an existing protocol for processing input, where invalid inputs are considered but still expected. In this case, unless *all* such inputs are formally specified as a part of the formal language of input, the rewriting is often co-opted by attackers [3]. As a result, more care should be put into what inputs should be allowed into a system, and by extension parsed and executed. Formally defining system inputs and rigorous testing are crucial steps in the direction of security assurance. Additionally, identifying and quantifying areas that break the paradigm of formal input verification on existing systems would be demonstrative of the scope of difficult for retrofitting existing systems into the LangSec realm.

2.2 Background

2.2.1 Weird Instructions and Weird Machines

Many computer programs accept and require input from sources that are unknown—such as untrusted users. To correctly process user input, a developer generates a series of rules (or *specification*) for how the inputs given to their program will be acted upon—including both processing of valid input and handling of unexpected input. If the specification is not carefully constructed to handle all possible input, an unwritten contract is forged between the program and user in which correct program behavior depends on correctly-formed input. In cases where this contract is not explicitly enforced, there are many opportunities to derail the intended operation of a program and guide it into states that were unexpected by the code's author, known as *weird instructions*. This type of data manipulation often requires a multi-step processes that can be spread throughout the execution timeline of a program to achieve a single goal [4].

A collection of weird instructions found inside of a program that when combined together creates an execution engine is known as a *weird machine.* These machines are unintended, emergent "virtual" machines that convert input data into execution flow through unexpected states and state transitions [5].

The *weird machine* model provides a formalized view of software exploitation. Return-oriented programming (ROP) is an example where a weird machine is constructed by mining an existing code base— commonly including the C standard library– for executable *gadgets* ending in RET instructions. Once a memory-corrupting bug is found capable of overwriting a program's stack frame, a chain of faked stack frames can act as a fully capable program executing on an emergent *weird machine* whose behavior is specified by the ROP-gadgets present. This form of execution directly parallels execution on a traditional computing machine: the addresses of these gadgets in memory form assembly-level bytecodes (op-codes) that drive execution of the weird machine to unintended states in the control flow graph (CFG). This construct has been described as early as 2001 by [6], [7], and its expressive power has been definitively described in terms of Turing-completeness in [8], finally updating the academic security's threat model from the narrowly-defined "malicious code" to the broader "malicious computation." [9]

2.2.2 Turing Completeness and Security Vulnerabilities

The fundamental notion that motivates the work of both Moka and Ristretto is that the prevalence of general-purpose, TC processors results in computing environments that provide excessive computational privilege. In many cases, execution environments provide significantly more capability than is necessary— capability that lies dormant until it is discovered and exploited by an attacker [10]. To understand why, a brief discussion of computational models is necessary.

The description of a machine's ability to process data and act upon it falls under the umbrella of computability theory. Various discrete classes of machine power are delineated by their ability to alter and recognize valid input data, i.e. accept inputs that are described by a grammar and reject all others. Critically, the languages that fall under a machine class share properties and behaviors; these features allow us to more easily reason about constituent programs. As an example, regular languages can be accepted by finite automata, whereas imperative languages with conditional branching require the full computational power of the Turing Machine.

Complex execution models—such as Turing machines—pose a challenging problem to LangSec-oriented development: due to the state space explosion and the undecidable nature of the halting problem, general-case formal verification of input acceptors is intractable. In recent research, the case has been made that TC languages—and their requisite TC runtime environments— are more powerful than most programs require, and that this extra power is a significant enabler for a compromising attacker [11].

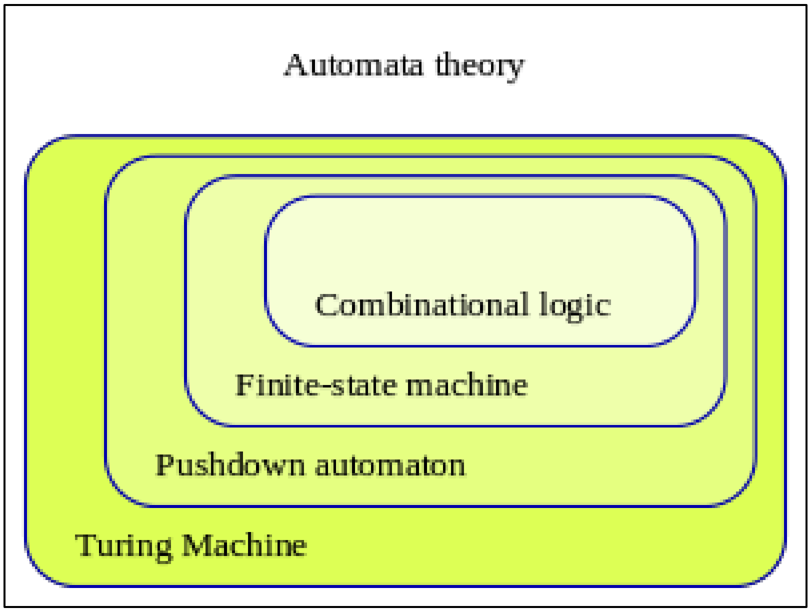

Figure 1: Computational Machines by Complexity Class

Traditionally, programming languages are designed to levy as much computational power as possible—facilitating use of the language to accomplish development tasks. LangSec research—which demonstrates the risks associated with unverified parsing code— suggests that the long-standing paradigm of computationally empowering developers is often a source of significant risk. In many cases, it may be desirable to provide a language that makes use of the least powerful, and most restricted execution model required to accomplish a given task. [2].

To this end, Moka inspected a large scale established code base to determine which percentage of code required the full power of a Turing machine. By inspecting the intermediate representation (IR) of code compilation— the internal language used by a compiler before translation into machine code— Moka identified regions of code that demonstrably do not require a full Turing Machine for execution. By doing so, the effort sought to demonstrate that the majority of code written by developers already exists in a sub-TC domain, and which consequently can be executed in a more constrained runtime environment, where formal analysis is a more tractable problem.

Ristretto makes use of the results provided by Moka, utilizing advanced compiler mechanisms to automatically convert code that does not require the full power of a Turing Complete language to a form that can be run in a computationally-restricted runtime environment. The is accomplished by removing unnecessary conditional branch instructions from the code by way of loop-unrolling.

Execution within a software-enforced computationally reduced environment significantly reduces a program's *state space*, reducing the number and size of any emergent weird machines, and increasing the ability of code analysis tools to achieve cover.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 The Moka Analysis System

3.1.1 Motivation

Past research efforts, such as Crema, have demonstrated the effectiveness of using restricted computation environments to improve analysis and verification of software—mitigating weird machines. The Moka effort sought to make the argument for the utility for the use of restricted computational environments by quantifying the prevalence of sub-TC semantics in a large and non-contrived codebase.

Of key interest in the Moka effort was the ability to determine whether a region of code could terminate. By demonstrating termination, the code can then be represented as a linear path of execution, and by extension, be represented by a forward-only executing machine. For example, a loop that has been shown to terminate can be unrolled such that instead of jumping back to a previous instruction, the code can be concatenated together (duplicated) the number of times the loop would execute.

While it is possible to manually classify the minimum machine that would recognize a section of code, and in some cases human analysis is necessary due to the Halting Problem, the need for repeatability and rapidity of analyzing large codebases led to the development of tools capable of automatically identifying many regions of sub-TC code. Moka uses the LLVM compiler's existing code-analysis engine and adds a series compilation passes. These passes operate on program semantics to discern the computational complexity required for each program function. The raw output from these passes is then processed with a series of Python scripts used for data gathering and analysis.

Moka's passes were developed to classify code into three categories: those that are trivially sub-TC, such as functions without any loops or recursive calls; functions that require the full power of the Turing machine; and functions that are more ambiguous to the LLVM compiler. The latter of these cases would require manual analysis and classification. From the onset, it was expected that regions requiring the full power of a Turing Machine would be found in the scheduler, anything requiring input handling, as well as the main architecture folder, as these would most likely have high numbers of spinlocks and indefinite looping constructs.

To make its argument, Moka used a minimal subset[1] of the Linux kernel (version 4.4.59), as a case study, and asked: what percentage of a large system can be executed in a restricted computational environment?

[1] Built with `make tinyconfig`

3.1.2 Moka Implementation

Moka was created to quantify the pervasiveness of code that could leverage the benefits of a restricted computational environment. To accomplish this, Moka extended the LLVM compiler with a series of passes which it used to gather data on a per-module basis by inspecting the intermediate representation (IR.) By doing so, it was possible to gather metrics concerning number of total functions, loop count, and source lines of code (SLOC) on a for the entirety of the Linux kernel. By performing these classifications, Moka identifies code that can be implemented in a sub-TC language such as Crema, and by extension, leverage the benefits of a restricted computational model. To contextualize its findings in terms of relative prevalence and/or importance, Moka focuses on two metrics:

1. Source Lines-of-Code (SLOC) – The Linux kernel has a well-defined source and by counting the number of SLOC (i.e., not comments or white-space) for transducers and full TC components as a percentage of the total, a metric for understanding the development burden to re-implement a project would be determined. This metric is valuable as future efforts may develop procedures to (semi-)automatically convert or constrain code, knowing a percentage of code available to convert would be desirable.

2. Loop Count – Examining only the SLOC however will not provide full insight into code regions worth-while to convert. The Linux source tree is very large, and highly configurable as to what code and functions are included in the compiled kernel, as plug-in modules or excluded. By identifying regions of code that have a large concentration of looping constructs, a development team can pinpoint the “low-hanging fruit” that is most exposed to possibly attacker-controlled input for the greatest security benefits.

3.1.3 Moka’s LLVM Passes and Utility Code

3.1.3.1 Cycle Pass

One of the factors for measuring if a function will terminate, and consequently be represented by a forward-only executing machine, is the existence of cycles in its function call graph (FCG) For example, in Figure 2 there are 3 functions:

```
#include <stdio.h>

void cycle_ref_1()
{
      cycle_ref_2();
      cycle_ref_3();
      return;
}

void cycle_ref_2()
{
      cycle_ref_1();
      return;
}

void cycle_ref_3()
{
      printf(“I am a terminating node of the FCG\n”);
      return;
}
```

Figure 2: Example of Call Cycle

In this contrived example, `cycle_ref_1` calls both `cycle_ref_2` and `cycle_ref_3`, and that `cycle_ref_2` will create a loop in the function call graph back into `cycle_ref_1`. However, due to the number of nodes in a call graph the in the Linux kernel, cycles are not so obvious, and the need for a programmatic approach is evident.

A pass was developed in the effort to identify these non-obviously-terminating sections that may exhibit this behavior, called a cycle pass. Like all other passes, the cycle pass is run on a per-file basis. Here, for each function found, cycle pass queries LLVM for all the child functions called by it. The output of this pass is returned to a corresponding python script that builds a graph of the function calls in the project directory. The script then performs a depth-first search on each parent function in the graph, and determines if any of the child nodes in the graph call back to the parent in question. If cycles exist in the call graph, the function and its corresponding cyclic call graph is added to a list, and saved in a JSON for later reporting.

```
{‘cycle_ref_2’: [‘cycle_ref_1’, ‘cycle_ref_2’],
 ‘cycle_ref_1’: [‘cycle_ref_2’, ‘cycle_ref_1’]}
```

Figure 3: The JSON output from the cycle pass has two cycles: cycle_ref_1 and cycle_ref_2.

3.1.3.2 Recursion Pass

Like cycles, functions containing direct recursive calls also make termination analysis a challenging problem for a machine to compute. To call attention to areas where this may be problematic, a recursion pass was written, and the identified code was manually analyzed for termination. The recursion pass operated on all the functions found in a module, and iterated over every block in the function. If a call instruction was identified, and the call was a function call with a name matching the parent function, the pass will add the recursive function identifying recursive calls to JSON file so that it can be manually analyzed for its ability to terminate

For example, the block of code seen in Figure 4 has two functions: `foo` which does nothing of note, and `recExample` which calls both `foo` and itself. The recursion pass lists all the call instructions found in the block (`foo` and `recExample`,) and checks them against the name of the enclosing function. In this instance, the pass will identify `recExample` as a recursive function and add it to the recursive calls JSON object.

```
void foo()
{
      while(0)
      {
            //do nothing
      }
}
void recExample()
{
      foo();
      recExample();
}
```

Figure 4: Recursive Function Example Code

3.1.3.3 Loop Count Pass

One of the major factors affecting the necessary computational power of a function is whether it has any loops that do not provably terminate on a finite input. A sub-TC machine can recognize and fully unwind a loop, and thus can be executed in a reduced power execution environment, and make use of models such as forward-only execution.

To this end, the Moka implements a series of checks on loop termination for a given function. For example, a simple loop:

```
for (i=0 ; i<10 ; i++)
```

when viewed in LLVM’s IR, has a clearly defined upper bound for the induction variable that can be simplified with built-in LLVM passes to determine its termination potential.

```
; <label>:3                        ; preds = %7, %0
      %4 = load i32, i32* %i, align 4
      %5 = icmp slt i32 %4, 10
      br i1 %5, label %6, label %10
```

Figure 5: LLVM IR for a simple loop

However, due to the complexity of the allowable syntax for loop headers, additional code was created to refactor ambiguous loop headers into ones that LLVM would be able to classify. For example:

```
int buff[5] = {0,1,2,3,4};
for (i = 0 ; i < buff[2] ; i++)
```

has an upper bound that isn’t resolved in the LLVM IR (%5)

```
; <label>:3                                    ; preds = %9, %0
  %4 = load i32, i32* %i, align 4
  %5 = getelementptr inbounds [5 x i32], [5 x i32]* %buff, i64 0, i64 2
  %6 = load i32, i32* %5, align 8
  %7 = icmp slt i32 %4, %6
  br i1 %7, label %8, label %12
```

Figure 6: Unresolved Upper Bound

Moka first runs a supplemental pass which allows for the resolution of variables that were previously undefined after compilation, called the argument promotion pass. This takes the `getelementptr` instruction, a method of resolving pointer values, and promotes it outside the loop, and thus allows LLVM to compute its value at compile-time (%5)

```
%4 = load i32, i32* %i, align 4
%5 = getelementptr inbounds [5 x i32], [5 x i32]* %buff, i64 0, i64 2
%6 = load i32, i32 * %4, align 8 ; :3
; <label>:3                                          ; preds = %9, %0
%7 = load i32, i32* %5, align 8
%8 = icmp slt i32 %4, %6
br i1 %8, label %9, label %12
```

Figure 7: Promotion Pass

3.1.3.4 Function Count

While not directly influencing the results of Moka, a metric that was used to put function termination into perspective was counting the raw number of functions found in the Linux kernel. This pass iterated over every LLVM bytecode file and listed all functions found. Coupled with a python script, this was used to verify function count results.

3.1.3.5 Source Lines of Code

One of the major metrics used for describing the relative concentration of Turing completeness in a file was defined by measuring the Source Lines of Code (SLOC) in a file and using that as a reference for scaling how impactful a non-terminating loop or function may be in the scope of an entire file or region of files. Since the Linux kernel organizes its files behaviorally, it is possible to derive some meaning from regions of concentration. Like the LLVM passes, a python script was created to iterate over every bytecode file, and run the Linux program, `SLOCCOUNT`, on the corresponding code file. SLOC for the languages found in each file was then saved was then saved in a conglomerated file for use by other programs.

3.1.3.6 Loop Graph

One of the core methods for representing the findings of Moka was showing the concentrations of loop non-termination in the folder topology—and by extension, functional blocks—of the Linux kernel, with the size being scaled by the total number of loops found. By doing so, it is possible to isolate regions where LangSec principles can be effectively applied and reimplemented in sub-TC languages such as Crema, and by contrast, areas where developers may have need for a Turing Complete execution environment and consequently require the use of a full TC language.

3.2 Ristretto

3.2.1 Motivation

To reduce the risk of software exploitation, Language- theoretic security (LangSec) provides two fundamental explanations of why exploits, or "weird machines" [5] crop up: *ad hoc* parsing of attacker-controlled data interleaving syntactic and semantic analyses, and overly-powerful programming/run-time environments. This work aims to bring a LangSec-inspired reduction of the computational "privilege" present in most current environments to bear with minimal overhaul of the development process.

If programs can be limited in their complexity and power, reference monitors and formal verification can improve the security when exposed to attacker-controlled input. Previous work [12] shows both empirical evidence of the improvement of verification as well as the broad applicability of such restricted computational environments. Presented herein is the design and initial evaluation of a system that can automatically provide hardware-enforcement of termination for program components identified to execute in a sub-Turing Complete space.

3.2.2 Implementation

To identify every loop at run-time as terminating or not is equivalent to the Halting Problem [13] — determining whether or not a general algorithm (equivalent to a Turing Machine) will halt on the given input. The scope of this effort is such that perfect coverage is not expected, instead Ristretto will focus on protecting the regions of a program that are provably sub-Turing complete

Compile Time Function Identification
Ristretto rapidly classifies a large number of components with an LLVM pass developed for Moka that marks every function as a candidate for run-time protections that either: does not contain any looping construct, or only contains looping constructs that fall within the bounds of Walter recursion — meaning termination is provable. These loops may already be candidates for unrolling during optimization or have fixed bounds. Those loops with defined bounds are marked for JIT unrolling, as in many cases the bounds are MAX_INT without additional run-time semantics.

JIT Loop Unrolling
During the execution of the protected application, when a region of the program that has been identified as provably bounded is reached, a run-time algorithm is used to determine a narrower

bound on loop iterations before unrolling the loop into machine code for that execution. At the boundaries of the unrolled region of machine code, a synchronization function is embedded to enable/disable the Processor Tracing (PT) monitor.

Within LLVM, this requires the “hoisting” of the loop into a separate function and module in order to prevent the JIT-compiled machine code from being cached and reused on future executions where the loop bounds may be different. In essence, the target loop is moved to an anonymous function in a temporary module, with the references to local and global variables updated to point to the context of the original function. At this point, the run-time loop bounds can be calculated using the *induction variable* and the values at loop entry. As the function identification step only marks loops that are known to terminate within at least MAX_INT iterations, the JIT engine will only operate on loops where the run-time information is sufficient to determine the back-edge count.

Forward-only Enforcement

A run-time mechanism implemented with Intel PT is used to identify deviations from expected execution flow and to terminate the offending process. Expected execution flow for many functions is determined by LLVM passes at compile time or with the JIT loop unroller. Unrolled functions can be expected to have a unidirectional execution flow and thus should not be the source of branches that decrease the instruction pointer. Such functions are framed with calls to an Intel PT library to trace the execution and parse the recorded execution flow. Return instructions that redirect execution to a caller at a lower address can also be accommodated by saving the expected return address when the trace starts.

Observation of the trace capture is currently done after a trace is completed. Deviations from expected program flow are caught and the process is terminated, however, in the case of a malicious attack this may be too late. Asynchronous observation of the trace capture would require relatively minimal additional engineering effort and provide real time protections against attacks. Doing so would involve moving the trace monitor to another thread (ideally its own CPU) and refactoring the parsing code to support real time parsing, i.e. parsing at a more granular level rather than requiring a complete parse.

Any attacks against a function protected by Ristretto must depend on forward-only branches that do not utilize return instructions (or sequences of instructions that perform similar semantics) in order to succeed, limiting the attacker to a weaker class of computational complexity. Ristretto provides the same level of power reduction on the attacker regardless of specific attack class including: ROP, stack-based buffer overflow, or even a code injection attack allowing the attacker to completely rewrite the code of a protected region.

4.0 RESULTS AND DISCUSSION

The results of Moka’s passes and analysis are outlined below, followed by the results of Ristretto. It should be noted that due to the complexity resulting from the full-TC language that the Linux kernel was implemented in, many of the loops required a sizeable amount of manual analysis to be performed.

4.1 Moka Pass Results

Following both: several loop header modification passes as well as manually analyzing the remaining loops that could not be programmatically classified, Moka found the vast majority of loops to be terminating, and thus able to run in a sub-TC execution environment.

Table 1: Loop Classification

Total Loops	2473
Terminating Loops	2180
Non-Terminating Loops	277
Non-Terminating Percent	11.2
Terminating Percent	88.2

By graphing the terminating loops against the number of loops in a file it is possible to determine areas in the kernel where Turing Completeness may be concentrated. Regions are color scaled towards the red where more non-terminating loops reside when compared to the rest of the kernel, and the size of the boxes depict the total loops found in them in comparison to the whole. Boxes that are found below headers in grey are subdirectories.

As seen in Figure 8, regions of high numbers of loop non-termination relative to the rest of the kernel can be found around the scheduler and to a lesser extent, the arch/x86, the drivers folder, and the event handler. Given that these are regions that require continuously and indefinitely iterating over values, these results are expected. It should be noted that these graphs do not represent the kernel in its entirety, but rather a high-level view of the top-level kernel directory.

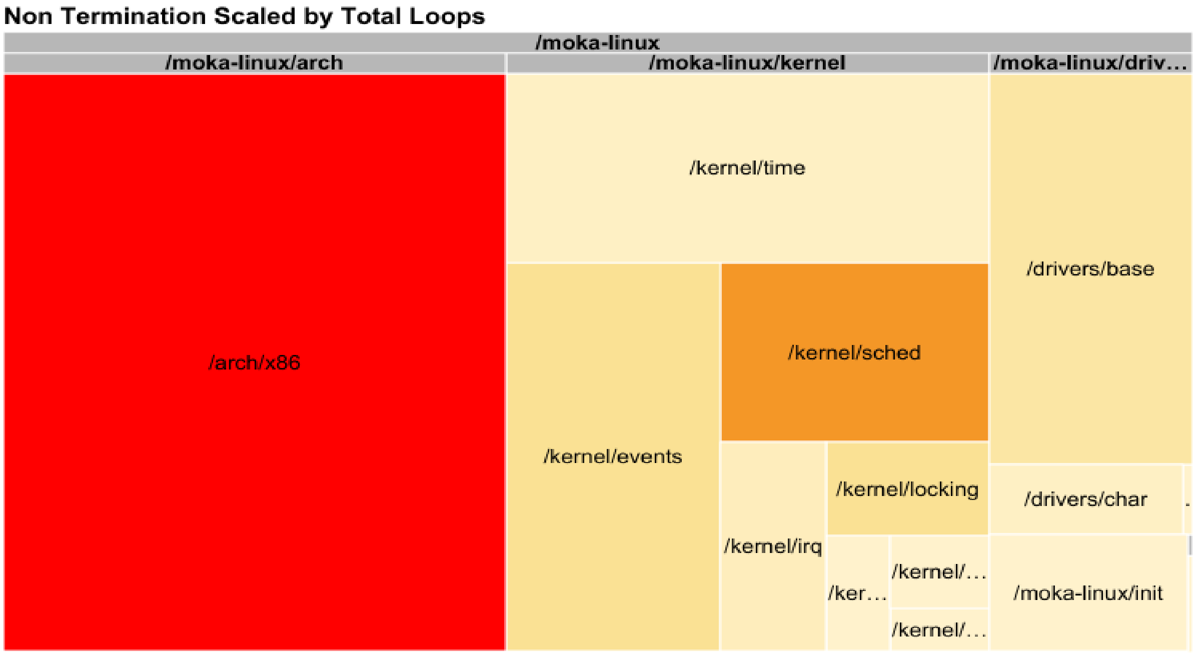

Figure 8: Graphical Representation of Non-Terminating Loops in Linux Kernel

Table 2: Notable Regions of Non-termination

kernel/sched	
Total loops	66
Non-terminating	47
arch/x86	
Total loops	396
Non-terminating	77
kernel/locking	
Total loops	21
Non-terminating	14
linux/mm	
Total loops	484
Non-terminating	50
linux/lib	
Total loops	391
Non-terminating	27

Additional metrics about the kernel that were found were two recursive calls, and a total of five cycles in function call graphs. Additionally, SLOC counts for languages used in the kernel were also gathered, which can be found in figure Table 3.

Table 3: Source Lines of Code by Source Language

SLOC Count Broken up by Language Type for the Linux Kernel	
ANISIC	**342092**
ASM	**9781**
CPP	**2460**

4.1.1 Interpretation of Results

As it has been shown, the clear majority of the code found by Moka using the Linux kernel can be run in a sub-TC execution environment. Consequently, most of the kernel could be reimplemented in a sub-TC language, such as Crema or have Crema-type termination checks on a subset of C, with core sections requiring the full computational power of a Turing Machine left in their original language.

During the scope of Moka's effort, some loop simplification code was developed to reduce the amount of code needing manual analysis. However due to the complexity of the allowable loop syntax, there remains much work to be done in the area. It would be possible to restrict the power of the computational machine executing sub-TC code, and thus garner the benefits of a LangSec environment, such as eliminating ROP attacks by enforcing a forward-only execution model.

4.2 Ristretto Results

4.2.1 Ropper

In order to evaluate the potential security improvements afforded by the Ristretto environment, an empirical analysis was performed on the impact of forward-only execution on an example attack class: ROP. The tool Ropper was used to find ROP chains in the target binaries to perform selected attacks. Ropper was then modified to only form chains of gadgets in ascending memory order—simulating execution in a forward-only environment. While there are weaknesses to this approach due to the loop unrolling performed in the JIT engine allowing for limited reuse of gadgets, most of the gadgets are selected from the libraries or other program functions and not within the function itself.

Ropper was tasked with finding a gadget chain for the following attacks:

- execve("/bin/sh") attack to execute a shell within the vulnerable process
- mprotect attack to weaken the memory protections for a Linux application
- VirtualProtect attack to weaken the memory protections for a Windows application

Each of these were tested in both 32-bit and 64-bit modes. For the Linux-based attacks, the following two targets were used:

- Statically-linked libc
- Dynamically-linked mysqld chosen due to its larger application size at between 200-245 MB depending on architecture

The Windows-based attack (VirtualProtect) was attempted against the below two targets:

- Microsoft’s cmd.exe
- Cygwin’s bash.exe

The initial findings show that in all cases, Ropper is able to find one or more valid ROP chain of gadgets within the target. When Ropper was restricted to only search for chains adhering to the forward-only model, it was *unable* to generate a chain for any attack × target × architecture. With this preliminary findings, we are optimistic that ROP-type attacks will be significantly hampered by the Ristretto compile and run-time protections in addition to other attack classes by fundamentally restricting their computational power.

4.2.2 Performance Impact

4.2.2.1 Intel Processor Tracing

In order to gain a sense of the performance overhead that the Intel PT monitoring thread will impose on the target application, a CPU-bound test program was developed to compute an approximation of π using an inefficient power-series definition (Figure 9). This program was run both with and without Intel PT monitoring its branching patterns, but not preventing backward jumps as the loop was not unrolled; the performance impact (Table 4) was approximately 5%. Each test was executed ten times with differing numbers of iterations to determine if there is a constant overhead for PT.

```
int iterations = 0, i;
float result = 0.0;
for (i = 0; i <= iterations; ++i)
{
 /∗ Pi approximation: Sum from i->infinity of
    ((-1)ˆi)/(1+(i∗2)) ∗/
 if (0 == (i%2)) // intentionally sloppy to slow up calculation
  {
   result += ((float)1)/((float)(1+(i∗2)));
      // casts keep the ints away
} else

  {
   result -= ((float)1)/((float)(1+(i∗2)));
} }
```

Figure 9: Power Series for Approximating Pi

Table 4: Intel PT Runtime Overhead

Iterations	**Time (w/o Trace)**	**Time (w/ Trace)**	**%Overhead**
900,000,000	5.2523s	5.5164s	5.02%
1,800,000,000	10.5108s	11.0167s	4.81%

4.2.2.2 Forward-Only Enforcement

Intel PT generates a stream of compressed data that represents the execution flow of the traced program. In order to detect backwards branches the captured data must be parsed by a decoder. Intel's XED decoder can decode approximately 2.2 million instructions per second [14]. This is miniscule compared to the number of instructions a modern CPU core can execute in the same time frame. In practice this may not be as significant as it first appears given that it's not pragmatic to enforce forward-only execution on a full program. The best use of the forward-only execution capability is to target code that handles program input. The relative size of such portions of code may differ greatly from one program to another. For many programs this will have a negligible impact for a typical end user, however the performance costs may become more pronounced in an enterprise environment. A more thorough study is necessary to understand the true impact of the parsing requirements.

5.0 CONCLUSION

By automatically identifying regions of code that are provably terminating and enforcing that termination bound, it is hypothesized that vulnerabilities could be mitigated. Through the use of a restricted execution model, software safety can be increased to reap the benefits of a LangSec-inspired approach without redesign or development. Future research and development aims to net performance and real-world security metrics for the consideration of the practical deployment of the Ristretto compiler tool-chain in a production environment. Other potential areas for future research include an exploration of different forward-only execution enforcement mechanisms (e.g. branch trapping) and subset processors that enforce restricted program behavior (minimal memory access, forward-only execution).

6.0 REFERENCES

[1] S. Bratus, M. Locasto, M. Patterson, L. Sassaman and A. Shubina, "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation," *;login:*, vol. 36, no. 6, pp. 13-21, Dec. 2011.

[2] L. Sassaman, M. Patterson, S. Bratus and M. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, pp. 489-500, 2013.

[3] E. Nava and D. Lindsay, "Abusing internet explorer 8's xss filters," Apr. 2010. [Online]. Available: http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf. [Accessed 12 Feb. 2015].

[4] S. Bratus, M. Locasto, M. Patterson, L. Sassaman and A. Shubina, "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," *;login*, pp. 13-21, December 2011.

[5] J. Vanegue, "The weird machines in proof-carrying code," in *Proc. First Annual Langsec Workshop*, 2014.

[6] Nergal, "The advanced return-into-lib(c) exploits," Dec. 2001. [Online]. Available: http://phrack.org/issues/58/4.html.

[7] G. Richarte, "Re: Future of buffer overflows," Oct. 2000. [Online]. Available: http://seclists.org/bugtraq/2000/Nov/32.

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[9] R. Hund, T. Holz and F. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th Conference on USENIX Security Symposium*, Berkeley, CA, 2009.

[10] S. Bratus, T. Darley, M. Locasto, M. L. Patterson, R. Shapiro and A. Shubina, "Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier," *IEEE Security & Privacy*, pp. 83-87, January/February 2014.

[11] L. Sassaman, M. Patterson, S. Bratus, M. Loscato and A. Shubina, "Security Applications of Formal Language Theory," Dartmouth College, Hanover, NH, 2011.

[12] J. I. Torrey and M. P. Bridgman, "Verification State-space Reduction through Restricted Parsing Environments," in *Security and Privacy Workshops*, San Jose, CA, 2015.

[13] R. M. a. J. U. J.E. Hopcroft, Introduction to Automata Theory, Languages, and Computation (3rd Edition), Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] B. Geßele, "GDSL Toolkit Intel Benchmarks," 16 April 2015. [Online]. Available: https://github.com/gdslang/gdsl-toolkit/wiki/IntelBenchmarks. [Accessed 30 May 2017].

[15] B. Cook, A. Podelski and A. Rybalchenko, "Termination proofs for system code," in *Proceedings of the 2006 ACM SIGPLAN conference*, New York, 2006.

[16] J. Mitolla, III, "Software radio architecture: a mathematical perspective," *IEEE Journal on Selected Areas in Communications* , pp. 514-538, 1999.

[17] S. T. Taft and F. Olsen, "Ada helps churn out less-buggy code," *Government Computer News*, pp. 2-3, June 1999.

[18] D. Turner, "Total Functional Programming," *Journal of Universal Computer Science. Vol. 10, No. 7.*, pp. 751-768, 2004.

[19] P. Wadler, "Comprehending Monads," in *Proceedings of the 1990 ACM conference on LISP and functional programming*, Nice, France, 1990.

[20] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*, Washington, DC, 2014.

[21] F. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, pp. 30-50, February 2000.

[22] U. Schoning, Theoretische Informatik - kurz gefaast. 5th ed., Heidelberg, Germany: Spektrum, 2008.

[23] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack*, pp. Vol. 7, 49:14, 11 August 1996.

[24] I. Ghory, "Using fizzbuzz to find developers who grok coding," Jan. 2007. [Online]. Available: http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/. [Accessed 20 Feb. 2015].

[25] C. Lattner, "The llvm compiler infrastructure," 2015. [Online]. Available: http://llvm.org. [Accessed 20 Feb. 2015].

[26] A. Lal, S. Qadeer and S. Lahiri, "Corral: A whole-program analyzer for boogie".

[27] G. Klein, "Operating system verification - an overview," *Sadhana,* vol. 34, no. 1, pp. 27-69, 2009.

[28] D. Bernstein, "qmail," 2013. [Online]. Available: http://cr.yp.to/qmail.html.

[29] S. Heelan, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities,* Oxford, UK: Master's thesis, University of Oxford, 2009.

[30] T. Avgerinos, "Automatic Exploit Generation," *Communications of the ACM,* vol. 57, no. 2, pp. 74-84, 2014.

[31] J. Vanegue, *The automated exploitation grand challenge*, H2HC, 2013.

[32] L. Sassaman, M. Patterson, S. Bratus and A. Shubina, "The halting problem of network stack insecurity," *USENIX ;login:,* vol. 36, no. 6, pp. 22-32, 2011.

[33] S. Bratus and F. Linder, "Information security war room," in *Proc. USENIX*, 2014.

[34] C. Walther, "Security applications of formal language theory," *Artificial Intelligence,* vol. 70, no. 1, 1994.

[35] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX OSDI 2008*, San Diego, CA, 2008.

[36] Microsoft Corporation, "Microsoft Research Boogie," 22 Oct. 2012. [Online]. Available: https://boogie.codeplex.com/. [Accessed 26 Feb. 2015].

[37] A. Lal, S. Qadeer and S. Lahiri, "Corral: A Whole-Program Analyzer for Boogie," in *First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, 2011.

[38] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto and A. Shubina, "Security Applications of Formal Language Theory," *IEEE Systems Journal*, vol. 7, no. 3, September 2013.

[39] A. Bogk and M. Schöpl, "The Pitfalls of Protocol Design: Attempting to Write a Formally Verified PDF Parser," in *Security and Privacy Workshops*, San Jose, CA, 2014.

7.0 LIST OF ACRONYMS

Abbreviation	**Full Name**
AST	Abstract Syntax Tree
AEG	Automatic exploit generation
BNF	Backus-Naur Form
CFG	Control flow graph
IR	Intermediate representation
JIT	Just-In-Time
LLVM	Low Level Virtual Machine
MTA	Mail transport agent
REPL	Read-evaluate-print loops
ROP	Return-oriented programming
stdlib	standard library
TC	Turing complete